Compilation of functional languages by program transformation

Pascal Fradet, Daniel Le Métayer

INRIA

UNITÉ DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.(1) 39 63 55 11

Rapports de Recherche

N° 1040

Programme 5

COMPILATION OF FUNCTIONAL LANGUAGES BY PROGRAM TRANSFORMATION

Pascal FRADET
Daniel LE METAYER

Mai 1989

RR-1040

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CÉDEX
FRANCE
Téléphone : 99 36 20 00
Télex : UNIRISA 950 473 F
Télécopie : 99 38 38 32

Compilation of Functional Languages by Program Transformation

Publication Interne n° 465 - Avril 1989 - 28 Pages

Pascal FRADET and Daniel LE METAYER

IRISA / INRIA
Campus de Beaulieu
35042 RENNES Cedex
FRANCE

Abstract

One of the most important issues concerning functional languages today is the efficiency and the correctness of their implementation. In this paper we describe the compilation process in terms of program transformations in the functional framework. The original functional expression is transformed into a functional term which can be seen as a traditional machine code. The two main steps are the compilation of the computation rule by the introduction of continuation functions and the compilation of the environment management using combinators. The advantage of this approach is that we do not have to introduce an abstract machine, which makes correctness proofs much simpler. As far as efficiency is concerned, this approach is promising since a lot of optimizations can be described and formally justified in the functional framework.

Compilation des langages fonctionnels par transformation de programmes

Résumé

Un des sujets de recherche majeurs dans le domaine des langages fonctionnels concerne leur mise en oeuvre et plus particulièrement deux points: l'efficacité et la correction. Dans ce rapport nous décrivons le processus de compilation en termes de transformations de programmes dans le cadre fonctionnel. L'expression fonctionnelle de départ est transformée en une expression fonctionnelle qui peut être considérée comme du code pour une machine traditionnelle. Les deux principales étapes sont la compilation du schéma d'évaluation à l'aide de fonctions "continuation" et la compilation de la gestion de l'environnement en introduisant des combinateurs. L'avantage de cette approche est d'éviter l'introduction d'une machine abstraite, ce qui simplifie les preuves de correction. L'approche est également prometteuse en ce qui concerne l'efficacité car beaucoup d'optimisations peuvent être décrites et justifiées formellement dans le cadre fonctionnel.

1. INTRODUCTION

The design of efficient and correct implementations is a crucial issue concerning functional langages today. As far as effiency is concerned, functional languages are still behind traditional imperative languages but much progress has been made within the last few years and the handicap is always lessening **[3,4,9,10]**. However functional languages possess a definite advantage over imperative languages: their semantic elegance makes program correctness proofs and program transformations easier to establish. In this paper we show that these properties can be turned into account to build a correct and efficient implementation based on program transformations.

The implementation of functional languages is generally described in terms of an abstract machine **[2,4,6,9,12]** reducing either the source functional program or a compiled version of this program. The abstract machine itself is implemented on a traditional von Neumann computer. So a complete correctness proof of the implementation should involve three steps:

(1) proof of the compilation process,

(2) proof that the reduction of a compiled expression by a specific computation rule and its execution on the abstract machine yield the same result **[14,15,17]**,

(3) proof that the implementation of the abstract machine is correct.

Part (1) is generally easy because the compilation produces a functional expression. Part (2) however involves the operational description of a specific machine which is much more difficult to tackle. Part (3) is generally omitted because the abstract machine is supposed to be close to the real one. This step would probably deserve more attention if the implementation of the abstract machine on the real one involves a non trivial translation process.

Since correctness proofs are much easier in the functional framework, we believe that the whole implementation process should be described in a purely functional way. We present a method for transforming λ-expressions into simpler functional expressions whose reduction can be seen as an execution on a traditional machine with two components: the code and the stack. *The important point is that we do not have to introduce a machine with an operational description indicating how the state evolves during the computation. The state of the "machine" is the expression itself and its evolution is specified by the definition of basic functions (combinators).* The functional expressions produced are of the form $F\ G\ S_1 \ldots S_n$ where F is a basic function which behaves like a machine instruction operating on the stack $S_1,\ldots,S_n$ and G is an expression representing the rest of the code.

This approach has interesting payoffs as far as correctness proofs are concerned. Let us notice that the code produced can be seen as generic code for a traditional machine which can be specialized for a particular processor by a straightforward macroexpansion; so we can say that we provide a proof of the whole implementation process. *Furthermore this approach allows the derivation of efficient code because various optimization techniques can be applied in a systematic*

way at each level of transformation. In our formalism these optimizations (some of them being well known by compiler constructors) are expressed as simple transformation rules which can easily be justified and systematically applied.

The execution of a functional program involves two main tasks:

(1) searching for the next expression to reduce according to a specific computation rule,

(2) management of the environment.

We achieve the compilation of these two tasks in the functional framework. Section 2 describes the compilation of the computation rule. The resulting expressions can be evaluated from left to right by successively applying the head function. We describe the compilation of call-by-value and call-by-name and give a sketch of the correctness proofs. Many improvements are possible on the expressions produced by this first step; section 3 introduces the most important ones and applies them to an example. In particular we describe a transformation rule corresponding to the well-known tail-recursion optimization. The compilation of environment management, presented in section 4, is done by an abstraction algorithm in the same spirit as **[13,21]**. This algorithm is based on a set of combinators acting like traditional machine instructions (move, push,...). A sketch of its correctness proof is given. This second transformation is independent of the computation rule; we present in section 5 various optimizations allowing a drastic improvement of the code; some of them are straightforward translations in the functional framework of well-known compiler optimization techniques such as peephole optimizations **[1]**. In conclusion we provide figures illustrating the efficiency of the produced code and we compare the method with previous work in two respects: the compiler derivation approach and the implementation choices.

2. COMPILATION OF THE COMPUTATION RULE

Our source language is a λ-calculus with constants described by the following syntax:

$$E ::= x \mid k \mid opm\ E_1 \ldots E_m \mid cond\ E_1\ E_2\ E_3 \mid E_1\ E_2 \mid \lambda x.E_1 \mid letrec\ f = \lambda x.E_1$$

where E_i are expressions, x is a variable, k is a basic constant, and opm is a strict primitive operator of arity m; the primitive cond is the only non strict operator.

The definition of factorial in this language is:

letrec fact = λx. cond (eq 0 x) 1 (mult x (fact (sub x 1)))

λ-calculus has been chosen because it is a simple language and it is sufficiently expressive to allow the translation of any high-level functional language into it. This means that if we have an implementation of the λ-calculus, we can implement any other functional languages by translating it into the λ-calculus.

We first consider a call-by-value semantics for the language. The compilation of call-by-name is given at the end of the section. The evaluation of an expression by call-by-value involves a repeated search for the next redex: for example the first operation to execute in the body of fact is eq, then

cond, then either 1 or sub, and so on ... The compilation of the computation rule should produce an expression which can be evaluated by systematic application of the first function (from left to right) (we could have chosen another particular function which can be found without recursive search, our choice results naturally from the analogy with machine code). Basically we have to invert the order of subexpressions in a composition: the call-by-value evaluation of a composition (E_1 E_2) entails the evaluation of E_2, then the evaluation of E_1, and finally the application of the result of E_1 to the result of E_2 (we take here the rightmost innermost interpretation of call-by-value: the leftmost innermost strategy could have been chosen as well). But replacing (E_1 E_2) by (E_2 E_1) is not correct ; we have to provide a mechanism for putting the result of E_2 back to its place after its evaluation. *This effect is achieved via the use of continuations; we transform each expression into an expression taking a continuation as argument and applying it to the result of its evaluation.* In the same way we define a new operator opm_c for each operator opm such that:

$opm_c\ C\ E_1 \ldots E_m = C\ (opm\ E_1 \ldots E_m)$ and $cond_c\ E_2\ E_3\ E_1 = cond\ E_1\ E_2\ E_3$

$cond_c$ is a particular function because it takes two possible continuations (E_2 and E_3).

Let us now take an example to illustrate the goal of the transformation described in this section. A call-by-value evaluation of the application (λx.add x (add x 1)) 3 would entail :

1) the searching for the next expression to reduce:	(λx.add x (add x 1)) 3
2) the β-reduction:	add 3 (add 3 1)
3) the searching for the next expression to reduce:	add 3 (add 3 1)
4) the first addition:	add 3 4
5) the second addition:	7

Our transformation (followed by a simple optimization that will be described later) applied to λx. add x (add x 1) will produce:

$\lambda x.\ add_c\ (add_c\ \mathbf{id}\ x)\ x\ 1$ where $\mathbf{id} = \lambda c.c$

In this expression (add_c **id** x) is the continuation of the leftmost add_c (it will by applied to its result) and **id** is the continuation of the rightmost add_c. The application of this new function to the value 3 can be evaluated by systematic reduction of the head function in the following way:

1) β-reduction:	add_c (add_c **id** 3) 3 1
2) first addition:	add_c **id** 3 4
3) second addition:	**id** 7
4) application of the continuation **id**:	7

This example shows in which sense the computation rule has indeed been compiled: there is no dynamic searching for the next redex during the evaluation.

Before giving a formal and comprehensive account of the transformation rules for the compilation of call-by-value, let us introduce them in a more intuitive way. We should keep in mind that the goal

of the transformation ($\mathbb{V}$ for call-by-value) is roughly to rearrange subexpressions according to the order in which they would be evaluated by call-by-value. To this aim the transformed expressions take an extra argument representing a continuation function to apply to the result of their evaluation (function calls will be of the form : F C $E_1 \ldots E_n$). Let us first consider expressions E which do not require any evaluation (i.e. weak head normal forms): these expressions are constants, variables (call-by-value indicates the evaluation of arguments before the function call) and λ-abstractions. The evaluation of these expressions consists only in returning their value, so they should be transformed into something like λc.c E' which means that the continuation can be applied at once. If E is a constant or a variable its value is obviously itself, so we have :

$\mathbb{V}(x) = \lambda c.c\ x$

$\mathbb{V}(k) = \lambda c.c\ k$

If E is a λ-abstraction λx.F, this expression itself should be transformed. It follows from the convention that any expression takes an extra argument that E' must be something like : $\lambda c.\lambda x.$ F' where F' represents the (transformed) expression F with continuation c; so we must have F' = $\mathbb{V}$(F) c and the rule for the λ-abstraction becomes:

$\mathbb{V}(\lambda x.F) = \lambda c.c\ (\lambda c.\lambda x.\mathbb{V}(F)\ c)$

The only case where some evaluation is required is the application. Let us first consider an expression $E = E_1\ E_2$ where E_1 is not a primitive operator. We should have as previously $\mathbb{V}(E_1\ E_2) = \lambda c.$ E'. Call-by-value indicates that E_2 should be evaluated first, so E' = $\mathbb{V}(E_2)$ E''. E'' is the continuation of $\mathbb{V}(E_2)$, so it represents the expression to evaluate after E_2. After the evaluation of E_2, call-by-value imposes the evaluation of E_1 and the application of the result of E_1 to the result of E_2; E_1 should return a function ; this function has to be applied to the value of E_2 with continuation c; so we must have E'' = $\mathbb{V}(E_1)$ **apply** c where **apply** f c = f c; so **apply** is equivalent to **id** in this context and we have:

$\mathbb{V}(E_1\ E_2) = \lambda c.\mathbb{V}(E_2)\ (\mathbb{V}(E_1)$ **id** c)

We can check that the rules for λ-abstraction and application are coherent. According the the previous rules we have:

$$\begin{aligned}\mathbb{V}((\lambda x.F)\ E) &= \lambda c.\mathbb{V}(E)\ (\mathbb{V}(\lambda x.F)\ \textbf{id}\ c)\\ &= \lambda c.\mathbb{V}(E)\ ((\lambda c.c\ (\lambda c.\lambda x.\ \mathbb{V}(F)\ c))\ \textbf{id}\ c)\\ &= \lambda c.\mathbb{V}(E)\ (\ (\lambda c.\lambda x.\ \mathbb{V}(F)\ c)\ \ c)\\ &= \lambda c.\mathbb{V}(E)\ (\ \lambda x.\ \mathbb{V}(F)\ c)\end{aligned}$$

So E is evaluated first, then λx.$\mathbb{V}$(F) c is applied to its result and $\mathbb{V}$(F) is evaluated with x bound to the value of E and c bound to the global continuation. Let us remark however that we must give the general rule for $\mathbb{V}(E_1\ E_2)$ because E_1 may not be a λ-abstraction : it may be itself an application or even a variable.

The transformation of a primitive operator can be explained along the same lines. In the expression opm $E_1 \ldots E_m$ call-by-value indicates the evaluation of E_m, E_{m-1},..., E_1 and then the

application of opm; so we have :

$$\mathbb{V}(\text{opm } E_1 \ldots E_m) = \lambda c.\mathbb{V}(E_m)\ (\mathbb{V}(E_{m-1})\ (\ldots(\mathbb{V}(E_1)\ (\text{opm}_c\ c))\ldots)\ )$$

For example $\mathbb{V}(\text{plus } E_1\ E_2) = \lambda c.\mathbb{V}(E_2)\ (\mathbb{V}(E_1)\ (\text{plus}_c\ c))$

$\mathbb{V}(E_1)\ (\text{plus}_c\ c)$ is the continuation of $\mathbb{V}(E_2)$ and $(\text{plus}_c\ c)$ is the continuation of $\mathbb{V}(E_1)$. So $\mathbb{V}(E_1)\ (\text{plus}_c\ c)$ is applied to N_2 (the result of E_2) giving $\mathbb{V}(E_1)\ (\text{plus}_c\ c)\ N_2$; $\text{plus}_c\ c$ is applied to N_1 (the result of E_1) and to N_2 giving $\text{plus}_c\ c\ N_1\ N_2$; the addition can be performed and the continuation is applied to N_1+N_2.

A special rule is attached to the conditional operator because it is strict in its first argument only. Futhermore cond takes two continuations and chooses among them according to the value of its argument. So in cond $E_1\ E_2\ E_3$, E_1 should be evaluated first, cond should then be applied to its result with two continuation corresponding to E_2 and E_3 ; E_2 and E_3 take the global continuation since they represent the result of the whole expression. So we should have:

$$\mathbb{V}(\text{cond } E_1\ E_2\ E_3) = \lambda c\ .\mathbb{V}(E_1)\ (\text{cond}_c\ (\mathbb{V}(E_2)\ c)\ (\mathbb{V}(E_3)\ c)\ )$$

The following figure gathers the transformation rules described above. The rule for letrec expressions can be deduced from the other rules and the definition of the call-by-value recursion combinator $\mathbf{Y_v}$ **[16]**: $\mathbf{Y_v} = \lambda f.\ (\lambda g.\ f\ (\lambda v.\ g\ g\ v))\ (\lambda g.\ f\ (\lambda v.\ g\ g\ v))$

($\mathbb{V}$**1**)	$\mathbb{V}(x)$	=	$\lambda c.c\ x$
($\mathbb{V}$**2**)	$\mathbb{V}(k)$	=	$\lambda c.c\ k$
($\mathbb{V}$**3**)	$\mathbb{V}(\text{opm } E_1 \ldots E_m)$	=	$\lambda c.\mathbb{V}(E_m)\ (\mathbb{V}(E_{m-1})\ (\ldots(\mathbb{V}(E_1)\ (\text{opm}_c\ c))\ldots)\ )$
($\mathbb{V}$**4**)	$\mathbb{V}(\text{cond } E_1\ E_2\ E_3)$	=	$\lambda c.\mathbb{V}(E_1)\ (\text{cond}_c\ (\mathbb{V}(E_2)\ c)\ (\mathbb{V}(E_3)\ c)\ )$
($\mathbb{V}$**5**)	$\mathbb{V}(E_1\ E_2)$	=	$\lambda c.\mathbb{V}(E_2)\ (\mathbb{V}(E_1)\ \mathbf{id}\ c)$
($\mathbb{V}$**6**)	$\mathbb{V}(\lambda x.E)$	=	$\lambda c.c\ (\lambda c.\lambda x.\ \mathbb{V}(E)\ c)$
($\mathbb{V}$**7**)	$\mathbb{V}(\text{letrec } f = \lambda x.E)$	=	$\lambda c.c\ (\text{letrec } f = \lambda c.\lambda x.\mathbb{V}(E)\ c)$

Let us come back to our simple introductory example to illustrate the transformation $\mathbb{V}$:

$\mathbb{V}(\lambda x.\text{add } x\ (\text{add } x\ 1))$

$= \lambda c.c\ (\lambda c.\lambda x.\mathbb{V}(\text{add } x\ (\text{add } x\ 1))\ c)$

$= \lambda c.c\ (\lambda c.\lambda x.(\lambda c.\mathbb{V}(\text{add } x\ 1)\ (\mathbb{V}(x)\ (\text{add}_c\ c)))\ c)$

$= \lambda c.c\ (\lambda c.\lambda x.(\lambda c.(\lambda c.\mathbb{V}(1)\ (\mathbb{V}(x)\ (\text{add}_c\ c)))\ (\mathbb{V}(x)\ (\text{add}_c\ c)))\ c)$

$= \lambda c.c\ (\lambda c.\lambda x.(\lambda c.(\lambda c.(\lambda c.c\ 1)\ ((\lambda c.c\ x)\ (\text{add}_c\ c)))\ ((\lambda c.c\ x)\ (\text{add}_c\ c)))\ c)$

We take the convention that top-level continuations are always **id**; exploiting this property and applying all possible β-reductions we get: $\lambda x.\text{add}_c\ (\text{add}_c\ \mathbf{id}\ x)\ x\ 1$

Section 3 contains a full treatment of optimization rules.

The evaluation described above shows that the transformed expression can be computed by

systematic reduction of the first function whithout dynamic search of the next redex.

In order to establish the correctness of this first transformation we have to show that the evaluation of the original expression by call-by-value (CBV) and the evaluation of the resulting expression by a computation rule modelling our intuitive notion of "systematic reduction of the first operator" (FIRST) yield the same result. To this aim we describe the two computation rules in the form of deductive systems in the style of **[18]**. The system is made of a set of axioms $E_1 \rightarrow_{CR} E_2$, which can be understood as CR (for Computational Rule) can reduce E_1 into E_2, and inference rules of the form:

$$\frac{E_1 \rightarrow_{CR} E'_1 \quad E_2 \rightarrow_{CR} E'_2}{exp(E_1,E_2) \rightarrow_{CR} exp(E'_1,E'_2)}$$

$exp(E_1,E_2)$ (resp. $exp(E'_1,E'_2)$) denoting an expression (possibly) involving E_1 and E_2 (resp. E'_1 and E'_2)

meaning that if E_1 can be reduced into E'_1 and E_2 can be reduced into E'_2 by CR then $exp(E_1,E_2)$ can be reduced into $exp(E'_1,E'_2)$ by CR. In formal terms such a system defines $\rightarrow_{CR}$ as the least relation containing the axioms and satisfying the inference rules.

In the following E_i denotes any expression in our language whereas N_i denotes weak head normal forms. In the remainder of the paper, the symbol $\equiv$ denotes a syntactic equality. We first give our definition of call-by-value:

Axioms:

opm $N_1 \ldots N_m$	$\rightarrow_{CBV}$	r	{result of application of opm to its arguments}
cond True E_1 E_2	$\rightarrow_{CBV}$	E_1	
cond False E_1 E_2	$\rightarrow_{CBV}$	E_2	
cond N E_1 E_2	$\rightarrow_{CBV}$	$\perp$	if $N \not\equiv$ True et $N \not\equiv$ False
$(\lambda x.\ E)\ N$	$\rightarrow_{CBV}$	E[N /x]	{E[N /x] denotes the expression Ewhere the occurences of x are replaced by N}
(letrec $f = \lambda x.E$) N	$\rightarrow_{CBV}$	E[(letrec $f = \lambda x.E$)/f][N/x]	
N E	$\rightarrow_{CBV}$	$\perp$	if $N \equiv k$, $N \equiv x$ or $N \equiv \perp$

Inference rules:

$$\frac{E_i \rightarrow_{cbv} E'_i}{\text{opm } E_1 \ldots E_i \; N_{i+1} \ldots N_m \rightarrow_{cbv} \text{opm } E_1 \ldots E'_i \; N_{i+1} \ldots N_m} \qquad \frac{E_i \rightarrow_{cbv} \perp}{\text{opm } E_1 \ldots E_i \; N_{i+1} \ldots N_m \rightarrow_{cbv} \perp}$$

$$\frac{E_1 \rightarrow_{cbv} E'_1}{\text{cond } E_1 \; E_2 \; E_3 \rightarrow_{cbv} \text{cond } E'_1 \; E_2 \; E_3} \qquad \frac{E_1 \rightarrow_{cbv} \perp}{\text{cond } E_1 \; E_2 \; E_3 \rightarrow_{cbv} \perp}$$

$$\frac{E_2 \rightarrow_{cbv} E'_2}{E_1 \; E_2 \rightarrow_{cbv} E_1 \; E'_2} \qquad \frac{E_2 \rightarrow_{cbv} \perp}{E_1 \; E_2 \rightarrow_{cbv} \perp}$$

$$\frac{E_1 \rightarrow_{cbv} E'_1}{E_1 \; N \rightarrow_{cbv} E'_1 \; N}$$

$\perp$ represents the undefined value; in order to deal with $\perp$, $\mathbb{V}$ can be extended with the rule $\mathbb{V}(\perp) = \perp$. We must introduce $\perp$ in order to be able to describe precisely the number of reduction steps required by a computational rule.

We describe now the system corresponding to the FIRST computation rule; this deductive system possesses a noteworthy property: *it is defined only in terms of axioms*. This is a formalization of the fact that FIRST involves no dynamic search of the next redex.

Axioms:

$\text{opm}_c \; C \; E_1 \ldots E_m \ldots E_n$	$\rightarrow_{FIRST}$	$C \; r \; E_{m+1} \ldots E_n$	with r the result of applying opm to $E_1..E_m$ and $r \neq \perp$
$\text{opm}_c \; C \; E_1 \ldots E_m \ldots E_n$	$\rightarrow_{FIRST}$	$\perp$	if $r \equiv \perp$
$\text{cond}_c \; E_1 \; E_2 \; \text{True} \; E_4 \ldots E_n$	$\rightarrow_{FIRST}$	$E_1 \; E_4 \ldots E_n$	
$\text{cond}_c \; E_1 \; E_2 \; \text{False} \; E_4 \ldots E_n$	$\rightarrow_{FIRST}$	$E_2 \; E_4 \ldots E_n$	
$\text{cond}_c \; E_1 \; E_2 \; E_3 \; E_4 \ldots E_n$	$\rightarrow_{FIRST}$	$\perp$	if $E_3 \not\equiv \text{True}$ and $E_3 \not\equiv \text{False}$
$(\lambda x. \; E_1) \; E_2 \; E_3 \ldots E_n$	$\rightarrow_{FIRST}$	$E_1[E_2/x] \; E_3 \ldots E_n$	
$(\text{letrec } f = \lambda x.E_1) \; E_2 \; E_3 \ldots E_n$	$\rightarrow_{FIRST}$	$E_1[(\text{letrec } f = \lambda x.E_1)/f][E_2/x] \; E_3 \ldots E_n$	
$E_1 \ldots E_n$	$\rightarrow_{FIRST}$	$\perp$	otherwise

For any computation rule CR (here CR = CBV or CR = FIRST) the result (normal form) of the evaluation of E by CR is denoted by CR(E); if the evaluation does not terminate CR(E) is

undefined. $N_{CR}(E)$ represents the number of reduction steps by CR to reach the normal form of E and $CR_1(E)$ is the expression obtained after one reduction step by CR. We first state two important lemmas relating the evaluation of $\mathfrak{V}(E)$ and $\mathfrak{V}(CBV_1(E))$ by FIRST.

Lemma 1:

$(\forall\ E, C)\ \ FIRST(\mathfrak{V}(E)\ C) \equiv FIRST(\mathfrak{V}(CBV_1(E))\ C)$

Lemma 2:

$(\forall\ E, C)$ if E is not in normal form then:

$N_{FIRST}(\mathfrak{V}(CBV_1(E))\ C) + 1 \leq N_{FIRST}(\mathfrak{V}(E)\ C) \leq N_{FIRST}(\mathfrak{V}(CBV_1(E))\ C) + k$

where k is a constant depending on the arities of primitive functions.

These two lemmas are proved by structural induction on the expressions; in fact we prove slightly generalized forms of these lemmas where the continuation C is replaced by a list of continuations $C_1 \ldots C_n$; this generalization is required by the induction itself. This proof is a routine inspection of the different cases [7]. To give its flavour we just describe the case $E = E_1\ E_2$ in the proof of lemma 1, where E_2 is not a normal form:

$FIRST(\mathfrak{V}(CBV_1(E_1\ E_2))\ C_1 \ldots C_n)$

$\equiv FIRST(\mathfrak{V}(E_1\ CBV_1(E_2))\ C_1 \ldots C_n)$ {definition of CBV_1}

$\equiv FIRST((\lambda c.\ \mathfrak{V}(CBV_1(E_2))\ (\mathfrak{V}(E_1)\ \mathbf{id}\ c))\ C_1 \ldots C_n)$ {definition of $\mathfrak{V}$}

$\equiv FIRST(\mathfrak{V}(CBV_1(E_2))\ (\mathfrak{V}(E1)\ \mathbf{id}\ C_1)\ C_2 \ldots C_n)$ {definition of FIRST}

$\equiv FIRST(\mathfrak{V}(E_2)\ (\mathfrak{V}(E_1)\ \mathbf{id}\ C_1)\ C_2 \ldots C_n)$ {induction hypothesis}

$FIRST(\mathfrak{V}(E_1\ E_2)\ C_1 \ldots C_n)$

$\equiv FIRST((\lambda c.\mathfrak{V}(E_2)\ (\mathfrak{V}(E_1)\ \mathbf{id}\ c))\ C_1 \ldots C_n)$ {definition of $\mathfrak{V}$}

$\equiv FIRST(\mathfrak{V}(E_2)\ (\mathfrak{V}(E_1)\ \mathbf{id}\ C_1)\ C_2 \ldots C_n)$ {definition of FIRST}

Lemma 2 allows us to derive a first correctness property:

Property 3:

The evaluation of E by CBV terminates if and only if there exists some C such that the evaluation of $\mathfrak{V}(E)\ C$ by FIRST terminates.

We can now introduce the general correctness property:

Property 4:

If the evaluation of E by CBV terminates then:

$(\forall\ E, C)\ \ FIRST(\mathfrak{V}(E)\ C) \equiv FIRST(\mathfrak{V}(CBV(E))\ C)$

Two interesting corollaries follow from this property:

Corollary 5: $(\forall\ E)\ CBV(E) \equiv k \Rightarrow FIRST(\mathfrak{V}(E)\ \mathbf{id}) \equiv k$

Corollary 6: $(\forall\ E)\ CBV(E) \equiv \lambda x.F \Rightarrow FIRST(\mathfrak{V}(E)\ \mathbf{id}) \equiv \lambda c.\lambda x.\mathfrak{V}(F)\ c$

Corollary 5 means that when the normal form of the expression by call-by-value is a basic value, the same result is obtained by the application of FIRST to the transformed expression with continuation **id**. Corollary 6 deals with expressions whose normal form is a λ-abstraction: in such cases the evaluation by FIRST of the transformed expression yields the corresponding function taking one extra argument representing its continuation.

The proof of property 4 is achieved by induction on $N_{CBV}(E)$:

- $N_{CBV}(E) = 0 \Rightarrow CBV(E) \equiv E$ and the property is trivially true.
- let us assume that the property holds for any E' such that $N_{CBV}(E') < N_{CBV}(E)$; then

$FIRST(\mathfrak{V}(CBV(E))\ C)$

$\equiv FIRST(\mathfrak{V}(CBV(CBV_1(E)))\ C)$ {definition of CBV_1}

$\equiv FIRST(\mathfrak{V}(CBV_1(E))\ C)$ {induction hypothesis}

$\equiv FIRST(\mathfrak{V}(E)\ C)$ {lemma 1}

This concludes the sketch of the correctness proof of the transformation $\mathfrak{V}$.

For the time being we have only described the compilation of the rightmost innermost computation rule. In order to show how other evaluation orders can be compiled by our method, we describe now the compilation of the leftmost innermost version of call-by-value and the compilation of call-by-name. The compilation of the leftmost innermost version of call-by-value is very similar to the previous one; the only differences concern rules ($\mathfrak{V}$**3**) and ($\mathfrak{V}$**7**) which become:

($\mathfrak{V}3_{ll}$) $\mathfrak{V}\ (opm\ E_1 \ldots E_n) = \lambda c.\ \mathfrak{V}(E_1)\ (\mathfrak{V}(E_2)\ (\ldots(\mathfrak{V}(E_m)(inv_m\ (opm_c\ c)))\ldots))$

where $inv_m\ C\ E_1 \ldots E_m = C\ E_m \ldots E_1$

($\mathfrak{V}7_{ll}$) $\mathfrak{V}\ (E_1\ E_2) = \lambda c.\ \mathfrak{V}(E_1)\ (\mathfrak{V}(E_2)\ (\lambda v_2.\ \lambda v_1.\ v_1\ c\ v_2))$

We give now the transformation rules corresponding to call-by-name:

($\mathfrak{N}$**1**) $\mathfrak{N}(x) = x$

($\mathfrak{N}$**2**) $\mathfrak{N}(k) = \lambda c.\ c\ k$

($\mathfrak{N}$**3**) $\mathfrak{N}(opm\ E_1 \ldots E_m) = \lambda c.\mathfrak{N}(E_m)\ (\mathfrak{N}(E_{m-1})\ (\ldots(\mathfrak{N}(E_1)\ (opm_c\ c))\ldots))$

($\mathfrak{N}$**4**) $\mathfrak{N}(cond\ E_1\ E_2\ E_3) = \lambda c.\mathfrak{N}(E_1)\ (cond_c\ (\mathfrak{N}(E_2)\ c)\ (\mathfrak{N}(E_3)\ c))$

($\mathfrak{N}$**5**) $\mathfrak{N}(E_1\ E_2) = \lambda c.\mathfrak{N}(E_1)\ \mathbf{id}\ c\ \mathfrak{N}(E_2)$

($\mathfrak{N}$**6**) $\mathfrak{N}(\lambda x.E) = \lambda c.c\ (\lambda c.\lambda x.\mathfrak{N}(E)\ c)$

($\mathfrak{N}$**7**) $\mathfrak{N}(letrec\ f = \lambda x.E) = letrec\ f = \lambda c.c\ (\lambda c.\lambda x.\mathfrak{N}(E)\ c)$

The two main departures from transformation $\mathfrak{V}$ appear in rules ($\mathfrak{N}$**1**) and ($\mathfrak{N}$**5**). By call-by-name arguments are not evaluated before a function call; this explains rule ($\mathfrak{N}$**1**): the argument is evaluated in the body of the function whereas it is just returned by call-by-value. Rule ($\mathfrak{N}$**5**) expresses the fact that the argument E_2 is passed unevaluated to the result of E_1 (since FIRST is intended to be the evaluation rule). We illustrate the definition of $\mathfrak{N}$ by the compilation of the simple expression that has already been used above for call-by-value:

$\mathfrak{N}$(λx. add x (add x 1))

= λc.c (λc.λx.$\mathfrak{N}$ (add x (add x 1)) c)

= λc.c (λc.λx. (λc.$\mathfrak{N}$ (add x 1) ($\mathfrak{N}$ (x) (add_c c))) c)

= λc.c (λc.λx. (λc.(λc. $\mathfrak{N}$ (1) ($\mathfrak{N}$ (x) (add_c c))) ($\mathfrak{N}$ (x) (add_c c))) c)

= λc.c (λc.λx. (λc.(λc. (λc.c 1) (x (add_c c))) (x (add_c c))) c)

Applying the same simplifications as above we get:

λx.x (add_c (x (add_c **id**))) 1

So the argument is evaluated first, then the first addition (with 1) can be computed; its continuation is (x (add_c **id**)), so x is reevaluated (if the arguments are shared it just returns the previously evaluated value) and the second addition can be performed. We do not dwell here on the correctness proof of the transformation $\mathfrak{N}$ since it is very similar to the proof of $\mathfrak{V}$; the interested reader may refer to [7] for a full description of the proofs.

3. SIMPLIFICATION RULES

As we already noticed, the expressions produced by the application of transformations $\mathfrak{V}$ and $\mathfrak{N}$ are rather bulky and may be simplified a lot by some trivial transformation rules. It is clear from the examples above that $\mathfrak{V}$ and $\mathfrak{N}$ introduce many new applications of λ-expressions which can be reduced by β-reduction. In order to ensure the termination of this reduction we apply the following strategy: a β-reduction is performed only if it strictly decreases the size of the expression. This restriction does not forbid most of the simplifications since all the compositions introduced by our transformations correspond to applications of an expression to its continuation and, by definition, a continuation is applied only once (at the end of the evaluation of the expression). We illustrate now this simplification with the compilation of the traditional recursive version of factorial by $\mathfrak{V}$:

$\mathfrak{V}$(letrec fact = λx.cond (eq 0 x) 1 (mult x (fact (sub x 1))))

= λc.c (letrec fact = λc.λx. $\mathfrak{V}$(cond (eq 0 x) 1 (mult x (fact (sub x 1)))) c)

= λc.c (letrec fact = λc.λx.$\mathfrak{V}$(eq 0 x) ($cond_c$ ($\mathfrak{V}$(1) c) ($\mathfrak{V}$(mult x (fact (sub x 1))) c)))

= λc.c (letrec fact = λc.λx.(λc.$\mathfrak{V}$(x) ($\mathfrak{V}$(0) (eq_c c))) ($cond_c$ ((λc.c 1) c)
((λc.$\mathfrak{V}$(fact (sub x 1)) ($\mathfrak{V}$(x) ($mult_c$ c))) c)))

= λc.c (letrec fact = λc.λx.(λc.(λc.c x) ((λc.c 0) (eq_c c))) ($cond_c$ ((λc.c 1) c)
((λc.(λc.$\mathfrak{V}$(sub x 1)($\mathfrak{V}$(fact) **id** c)) ((λc.c x) ($mult_c$ c))) c)))

$= \lambda c.c$ (letrec fact = $\lambda c.\ \lambda x.(\lambda c.(\lambda c.c\ x)\ ((\lambda c.c\ 0)\ (eq_c\ c)))\ (cond_c\ ((\lambda c.c\ 1)\ c)$
$((\lambda c.(\lambda c.(\lambda c.\Psi(x)\ (\Psi(1)\ (sub_c\ c)))((\lambda c.c\ fact)\ \mathbf{id}\ c))$
$((\lambda c.c\ x)\ (mult_c\ c)))\ c)))$

$= \lambda c.c$ (letrec fact = $\lambda c.\ \lambda x.(\lambda c.(\lambda c.c\ x)\ ((\lambda c.c\ 0)\ (eq_c\ c)))\ (cond_c\ ((\lambda c.c\ 1)\ c)$
$((\lambda c.(\lambda c.(\lambda c.(\lambda c.c\ x)\ ((\lambda c.c\ 1)\ (sub_c\ c)))((\lambda c.c\ fact)\ \mathbf{id}\ c))$
$((\lambda c.c\ x)\ (mult_c\ c)))\ c)))$

This expression can be simplified by a succession of β-reductions into:

$\lambda c.c$ (letrec fact = $\lambda c.\lambda x.eq_c\ (cond_c\ (c\ 1)\ (sub_c\ (fact\ (mult_c\ c\ x))\ x\ 1))\ 0\ x)$

We have already mentioned that top-level continuations are assumed to be equal to **id**. This convention is rather natural since top-level expressions are just supposed to return their result. This property can be exploited to achieve new improvements of the code. For example, the result of the compilation of fact above can be transformed into:

letrec fact = $\lambda c.\ \lambda x.\ eq_c\ (cond_c\ (c\ 1)\ (sub_c\ (fact\ (mult_c\ c\ x))\ x\ 1))\ 0\ x$

In order to convince the reader that the functions produced by our transformations can really be evaluated by reducing systematically the first operator of the current expression, we describe now the evaluation of (fact **id** 1). The operator applied at each step is underlined.

(letrec fact = $\lambda c.\ \lambda x.\ eq_c\ (cond_c\ (c\ 1)\ (sub_c\ (fact\ (mult_c\ c\ x))\ x\ 1))\ 0\ x)$ **id** 1

$\rightarrow_{FIRST}$ $(\underline{\lambda x}.eq_c\ (cond_c\ (\mathbf{id}\ 1)\ (sub_c\ (fact\ (mult_c\ \mathbf{id}\ x))\ 1\ 1))\ 0\ x)\ 1$

$\rightarrow_{FIRST}$ $\underline{eq_c}\ (cond_c\ (\mathbf{id}\ 1)\ (sub_c\ (fact\ (mult_c\ \mathbf{id}\ 1))\ 1\ 1))\ 0\ 1$

$\rightarrow_{FIRST}$ $\underline{cond_c}\ (\mathbf{id}\ 1)\ (sub_c\ (fact\ (mult_c\ \mathbf{id}\ 1))\ 11)$ False

$\rightarrow_{FIRST}$ $\underline{sub_c}\ (fact\ (mult_c\ \mathbf{id}\ 1))\ 1\ 1$

$\rightarrow_{FIRST}$ $\underline{(letrec}$ fact = $\lambda c.\lambda x.eq_c\ (cond_c\ (c\ 1)\ (sub_c\ (fact\ (mult_c\ c\ x))\ x\ 1))\ 0\ x)\ (mult_c\ \mathbf{id}\ 1)\ 0$

$\rightarrow_{FIRST}$ $(\underline{\lambda x}.eq_c\ (cond_c\ (\ mult_c\ \mathbf{id}\ 1\ 1)\ (sub_c\ (fact\ (mult_c\ (mult_c\ \mathbf{id}\ 1)\ x))\ x\ 1))\ 0\ x)\ 0$

$\rightarrow_{FIRST}$ $\underline{eq_c}\ (cond_c\ (\ mult_c\ \mathbf{id}\ 1\ 1)\ (sub_c\ (fact\ (mult_c\ (mult_c\ \mathbf{id}\ 1)\ 0\ ))\ 0\ 1))\ 0\ 0$

$\rightarrow_{FIRST}$ $\underline{cond_c}\ (\ mult_c\ \mathbf{id}\ 1\ 1)\ (sub_c\ (fact\ (mult_c\ (mult_c\ \mathbf{id}\ 1)\ 0\ ))\ 0\ 1))$ True

$\rightarrow_{FIRST}$ $\underline{mult_c}\ \mathbf{id}\ 1\ 1$

$\rightarrow_{FIRST}$ $\underline{\mathbf{id}}\ 1$

$\rightarrow_{FIRST}$ 1

Let us now introduce another straightforward optimization; the transformation of the expression

$(\lambda x.\lambda y.\ E)\ X\ Y$

by Ψ yields after simplification:

$\lambda c.\Psi(Y)\ (\Psi(X)\ (\lambda x.(\lambda c.\lambda y.\ \Psi(E)\ c))\ c)$

No β-reduction is possible on this expression; however it is clear that the continuation of $\Psi(E)$ will be the global continuation (i.e.the rightmost occurrence of c). So the result we would like is:

$\lambda c.\mathbb{V}(Y)\ (\mathbb{V}(X)\ (\lambda x.(\lambda y.\mathbb{V}(E)\ c)))$

This optimization can be obtained by the introduction of new rules in the definition of $\mathbb{V}$:

$(\mathbb{V}5_l)\ \mathbb{V}((\lambda x_1 ... \lambda x_n.\ F)\ E_1\ ...\ E_n) = \lambda c.\mathbb{V}(E_n)\ (\ ...\ (\mathbb{V}(E_1)\ (\lambda x_1 ... \lambda x_n.\mathbb{V}(F)\ c))\ ...\ )$

This rule is very similar to the rule ($\mathbb{V}3$) for the complete application of a primitive operator to its arguments. The same technique can be used (with certain restrictions) when the function is defined by a letrec expression. To illustrate these new definitions, we take the example of an iterative version of factorial:

letrec fact' = $\lambda x.\ \lambda y.$ cond (eq 0 x) y (fact' (sub x 1) (mult x y))

Applying the optimizations described above, this function is transformed into:

letrec fact' = $\lambda c.\ \lambda x\ .\ \lambda y.\ eq_c\ (cond_c\ (c\ y)\ (mult_c\ (sub_c\ (fact'\ c)\ x\ 1)\ x\ y))\ 0\ x$

This example allows us to show how the traditional tail-recursion optimization can be expressed in our framework. In operational terms a tail-recursive function is a function such that no computation has to be carried out after the recursive calls; in functional terms this means that the continuation is always the identity function and the tail-recursion optimization consists in removing this identity continuation, that is to say producing a function taking no continuation. Let us define function fact'' in the following way:

letrec fact'' = fact' **id**

using the definition of fact', we get:

letrec fact'' = $\lambda x.\lambda y.eq_c\ (cond_c\ (c\ y)\ (mult_c\ (sub_c\ (fact'$ **id**$)\ x\ 1)\ x\ y))\ 0\ x$

which can be transformed into (folding the definition of fact''):

letrec fact'' = $\lambda x.\lambda y.eq_c\ (cond_c\ (c\ y)\ (mult_c\ (sub_c\ fact''\ x\ 1)\ x\ y))\ 0\ x$

We have obtained a new function which does not take any continuation; this transformation is possible because all the recursive calls to fact' in (fact' **id**) take **id** as a continuation (otherwise the final folding would have been impossible). So we have expressed in the functional framework the definition of tail-recursion and the corresponding optimization. *It is an important payoff of the functional approach that most well-known compiler optimization techniques can be expressed (and formally justified) by simple program transformation rules.*

4. COMPILATION OF ENVIRONMENT MANAGEMENT

Let us come back to the evaluation of (fact **id** 1) described in the previous section to make a comparison with machine code execution. Throughout the reduction of (fact **id** 1) the expression under evaluation is always of the form: $E_1\ E_2\ E_3\ ...\ E_n$ where E_1 is the next function to apply and E_2 its continuation. When E_1 is evaluated and returns the value N_1, the expression becomes $E_2\ N_1\ E_3\ ...E_n$. Let us now look at the expression under evaluation as a machine state. Clearly E_1 would be the next instruction to execute, E_2 would be the rest of the code and $E_3,...,E_n$ would play the role of a stack. However these expressions are still far from machine code; the basic reason is

the occurrence of λ-expressions whose reduction is not straightforward: it involves some kind of environment management. *We use a well-known technique for the compilation of environment management within the functional framework, which is called abstraction [5,13,21].* The abstraction process consists in translating a functional expression into an equivalent one which contains no variables via the use of combinators. Combinators-based compilers generally use combinators such as **S**, **K**, **I** [21] or indexed combinators [13]. These combinators must be then implemented on a real machine: they may be interpreted or custom hardware may be designed to support them. We take here an alternative approach: *we choose a set of indexed combinators which act on their arguments as traditional machine instructions on a stack:*

$$\begin{aligned}
&\mathbf{push}\ E\ F && = F\ E && \\
&\mathbf{dupl_n}\ E\ S_1 \ldots S_n && = E\ S_n\ S_1 \ldots S_n && \\
&\mathbf{flsh_n}\ E\ S_1 \ldots S_n && = E && \\
&\mathbf{move_{m,n}}\ E\ S_1 \ldots S_{m-1}\ S_m\ S_{m+1} \ldots S_n && = E\ S_1 \ldots S_{m-1}\ S_n\ S_{m+1} \ldots S_n && \text{if } n \geq m \\
&\mathbf{move_{m,n}}\ E\ S_1 \ldots S_n \ldots S_{m-1}\ S_m && = E\ S_1 \ldots S_n \ldots S_{m-1}\ S_n && \text{if } n < m \\
&\mathbf{ldcl_n}\ E\ F\ S_1 \ldots S_n && = E\ (F\ S_n)\ S_1 \ldots S_n &&
\end{aligned}$$

These combinators can be seen as machine instructions operating in the following way:

- **id** corresponds to a return instruction: if the stack contains a single element then **id** returns it ; otherwise it causes a jump to the address given by the top of the stack,
- **push** is the traditional push instruction with an immediate argument,
  (**push** $E\ F\ S_1 \ldots S_n = F\ E\ S_1 \ldots S_n$)
- $\mathbf{dupl_n}$ pushes the n^{th} element of the stack,
- $\mathbf{move_{m,n}}$ replaces the m^{th} element of the stack by the n^{th} element,
- $\mathbf{flsh_n}$ pops the first n elements of the stack: it amounts to a modification of the stack pointer,
- $\mathbf{ldcl_n}$ is used to build a closure on the top of the stack. In a real machine the top of the stack would contain a pointer to the currently built closure and $\mathbf{ldcl_n}$ would involve the allocation of a new memory cell to hold the new value.

In order to make the description of the abstraction algorithm clearer, we introduce more powerful combinators which can be defined in terms of the previous ones:

$$\begin{aligned}
&\mathbf{delt_{m,n}}\ E\ P_1 \ldots P_m\ X_1 \ldots X_n && = E\ P_1 \ldots P_m \\
&\mathbf{exed_{m,n,i}}\ P_1 \ldots P_m\ X_1 \ldots X_n && = X_i\ P_1 \ldots P_m \\
&\mathbf{mkcl_{m,n}}\ E\ F\ P_1 \ldots P_m\ X_1 \ldots X_n && = E\ (F\ X_1 \ldots X_n)\ P_1 \ldots P_m\ X_1 \ldots X_n
\end{aligned}$$

In operational terms $\mathbf{exed_{m,n,i}}$ is a jump to the address contained in the i+mth element of the stack after having removed from the stack the n elements corresponding to the current "environment" (i.e. arguments of the function under evaluation). The first m elements correspond to the arguments of the called function. The combinator $\mathbf{mkcl_{m,n}}$ builds in one step a closure on the top of the stack and $\mathbf{delt_{m,n}}$ removes n elements from the stack. The following properties can be easily checked:

$\mathbf{delt_{m,n}}\ E \quad = \quad \mathbf{move_{n+m,m}}\ (\ldots(\mathbf{move_{n+1,1}}\ (\mathbf{flsh_n}\ E))\ldots)$

$\mathbf{exed_{m,n,i}} \quad = \quad \mathbf{dupl_{m+i}}\ (\mathbf{delt_{m+1,n}}\ \mathbf{id})$

$\mathbf{mkcl_{m,n}}\ E \quad = \quad \mathbf{ldcl_{m+1}}\ (\ldots(\mathbf{ldcl_{m+n}}\ E)\ldots)$

We can now present our abstraction algorithm ; the abstraction of variables $x_1,\ldots,x_n$ from M is denoted by $[x_1,\ldots,x_n]_0$ M. In other words $[x_1,\ldots,x_n]_0$ M is an expression which does not contain $x_1,\ldots,x_n$ such that $([x_1,\ldots,x_n]_0\ M)\ x_1\ldots x_n = M$. Actually we give a more general definition of the abstraction $[x_1,\ldots,x_n]_p$ M such that:

$(\forall\ S_1,\ldots,S_p) \qquad ([x_1,\ldots,x_n]_p\ M)\ S_1\ldots S_p\ x_1\ldots x_n = M\ S_1\ldots S_p$

We take the convention that when a function is called its arguments are on the top of the stack. The execution of a function may involve the installation of new elements on the top of the stack: index p in the abstraction algorithm denotes the number of values pushed on the stack over the arguments of the function at a particular execution step. So the ith argument of a function can always be found at the position p+i in the stack.

The global expression is first normalized: nested λ-abstractions are transformed into closed λ-expressions in the following way:

$\lambda x_1.\ \ldots\lambda x_n.\ exp$ --> $(\lambda y_1.\ \ldots\lambda y_k.\lambda x_1.\ \ldots\lambda x_n.\ exp)\ y_1\ \ldots\ y_k$

where y_1 , ..., y_k are the free variables of the original λ-expression.

This normalization, which is very much in the spirit of supercombinators **[8]** (but does not exhibit full laziness) allows us to apply the abstraction algorithm to the innermost λ-expressions in a bottom-up fashion. The basic abstraction rules are defined in the following way (when the left hand sides of two rules overlap, the first rule should be applied first) :

(A1).	$[x_1,\ldots,x_n]_p\ M$	$= \mathbf{delt_{p,n}}\ M$	if $x_1,\ldots,x_n \notin M$ and $n \neq 0$
(A2).	$[x_1,\ldots,x_n]_p\ M\ x_i$	$= \mathbf{dupl_{p+i}}\ ([x_1,\ldots,x_n]_{p+1}\ M)$	if $x_i \in \{x_1,\ldots,x_n\}$
(A3).	$[x_1,\ldots,x_n]_p\ M\ y$	$= ([x_1,\ldots,x_n]_{p+1}\ M)\ y$	if $y \notin \{x_1,\ldots,x_n\}$
(A4).	$[x_1,\ldots,x_n]_p\ M\ k$	$= \mathbf{push}\ k\ ([x_1,\ldots,x_n]_{p+1}\ M)$	
(A5).	$[x_1,\ldots,x_n]_p\ opm_c\ M$	$= opm_c\ ([x_1,\ldots,x_n]_{p-m+1}\ M)$	$(p \geq m)$
(A6).	$[x_1,\ldots,x_n]_p\ cond_c\ M\ N$	$= cond_c\ ([x_1,\ldots,x_n]_{p-1}\ M)\ ([x_1,\ldots,x_n]_{p-1}\ N)$	$(p \geq 1)$
(A7).	$[x_1,\ldots,x_n]_p\ x_i$	$= \mathbf{exed_{p,n,i}}$	if $x_i \in \{x_1,\ldots,x_n\}$
(A8).	$[x_1,\ldots,x_n]_p\ M\ N$	$= \mathbf{push}\ ([x_1,\ldots,x_n]_0\ N)\ (\mathbf{mkcl_{p,n}}\ ([x_1,\ldots,x_n]_{p+1}\ M))$	if M is not a basic operator

Before tackling the correctness proof we give an intuitive justification for these rules:

(A1) means that the arguments of the function can be discarded from the stack as soon as they are no longer referenced in the remaining code.

(A2) indicates that a composition $(M\ x_i)$ is evaluated by first pushing the value of x_i (which can be found at position p+i in the stack) and then evaluating M with one more element on the stack.

(A4) achieves the same effect with a constant argument k.

(A3) is applied in the abstraction of a nested expression containing free variables (function names) which are left unchanged.

(A5) describes the treatment of operators; an operator of arity m consumes m elements from the stack and produces one element; so the remainder of the expression is compiled with index p-m+1.

(A6) expresses the fact that $cond_c$ consumes one (boolean) element and then transfers the control to one of its alternatives.

(A7) is applied when the remaining expression is an argument which means that all other arguments can be discarded. This effect is achieved by the **exed** combinator.

(A8) deals with the evaluation of a non-basic expression M with a non basic continuation N. A representation of the continuation N must be pushed on the stack with its environment so that the expression M can call it after its own evaluation (this is achieved by **push** and **mkcl**).

Let us now come back to our simple example to illustrate this abstraction algorithm; we have described in section 3 the transformation of the expression $(\lambda x.\ add\ x\ (add\ x\ 1))$ by $\mathfrak{V}$; the result was (after simplification):

$$\lambda x.\ add_c\ (add_c\ \mathbf{id}\ x)\ x\ 1$$

The application of our abstraction algorithm to this expression gives:

$[x]_0\ (add_c\ (add_c\ \mathbf{id}\ x)\ x\ 1)$

$= \mathbf{push}\ 1\ ([x]_1\ (add_c\ (add_c\ \mathbf{id}\ x)\ x\ ))$ **(A4)**

$= \mathbf{push}\ 1\ (\mathbf{dupl_2}\ ([x]_2\ (add_c\ (add_c\ \mathbf{id}\ x))))$ **(A2)**

$= \mathbf{push}\ 1\ (\mathbf{dupl_2}\ (add_c\ ([x]_1\ (add_c\ \mathbf{id}\ x))))$ **(A5)**

$= \mathbf{push}\ 1\ (\mathbf{dupl_2}\ (add_c\ (\mathbf{dupl_2}\ ([x]_2\ (add_c\ \mathbf{id})))))$ **(A2)**

$= \mathbf{push}\ 1\ (\mathbf{dupl_2}\ (add_c\ (\mathbf{dupl_2}\ (\mathbf{delt_{2,1}}\ (add_c\ \mathbf{id}))))$ **(A1)**

We can now state the correctness property of our abstraction algorithm:

Property 7:

$(\forall\ p \geq 0)\ (\forall\ M, S_1, \ldots, S_p)\quad ([x_1,\ldots,x_n]_p\ M)\ S_1 \ldots S_p\ x_1 \ldots x_n = M\ S_1 \ldots S_p$

This property can be shown by induction on the structure of the expressions that can be produced by the first transformation. The proof is a routine inspection of the different cases; let us just describe the case $E \equiv M\ N$:

$([x_1,\ldots,x_n]_p\ M\ N)\ S_1\ldots S_p\ x_1\ldots x_n$

$= \mathbf{push}\ ([x_1,\ldots,x_n]_0\ N)\ (\mathbf{mkcl_{p,n}}\ ([x_1,\ldots,x_n]_{p+1}\ M))\ S_1\ldots S_p\ x_1\ldots x_n$ **(A8)**

$= \mathbf{mkcl_{p,n}}\ ([x_1,\ldots,x_n]_{p+1}\ M)\ ([x_1,\ldots,x_n]_0\ N)\ S_1\ldots S_p\ x_1\ldots x_n$ {definition of push}

$= ([x_1,\ldots,x_n]_{p+1}\ M)\ (([x_1,\ldots,x_n]_0\ N)\ x_1\ldots x_n)\ S_1\ldots S_p\ x_1\ldots x_n$ {definition of mkcl}

$= M\ (([x_1,\ldots,x_n]_0\ N)\ x_1\ldots x_n)\ S_1\ldots S_p$ {induction hypothesis}

$= M\ N\ S_1\ldots S_p$ {induction hypothesis}

5. OPTIMIZATION TECHNIQUES

The abstraction algorithm given in section 4 is rather straightforward and the code produced can be improved in several ways. We first describe some optimizations of the algorithm itself, then we present a simplification process which can be applied to the result of the abstraction algorithm.

It is well known that the manipulation of closures is an important source of inefficiency in the implementation of functional languages; so closures should be built only when necessary and the unavoidable closures should be kept as small as possible. Rule (**A8**) can be improved in order to gather in the closure only the values required for the evaluation of the expression:

(**A8**). $[x_1,\ldots,x_n]_p\ M\ N = \mathbf{push}\ ([x_{i1},\ldots,x_{ir}]_0\ N)\ (\mathbf{ldcl_{p+i1}}\ldots(\ \mathbf{ldcl_{p+ir}}\ ([x_1,\ldots,x_n]_{p+1}\ M))\ldots)$
where $x_{i1},\ldots,x_{ir}$ are the free variables of N.

In order to describe the second optimization of the algorithm we must introduce some definitions:

Definition 8:

We consider another abstraction algorithm $[x_1,\ldots,x_n]'_p\ M$ very similar to the previous one (replacing $[x_1,\ldots,x_n]_p$ by $[x_1,\ldots,x_n]'_p$) except that (**A1**), (**A7**), (**A8**) become:

(**A1'**) $[x_1,\ldots,x_n]'_p\ k = k$

(**A7'**) $[x_1,\ldots,x_n]'_p\ x_i = \mathbf{dupl_{p+i}}\ \mathbf{id}$

(**A8'**) $[x_1,\ldots,x_n]'_p\ M\ N = \mathbf{push}\ ([x_1,\ldots,x_n]_0\ N)\ (\mathbf{mkcl_{p,n}}\ ([x_1,\ldots,x_n]'_{p+1}\ M))$

The interest of this new algorithm appears in the following property:

Property 9:

$\forall\ p \geq 0\ \ \forall\ M, S_1,\ldots,\ S_p\quad ([x_1,\ldots,x_n]'_p\ M)\ S_1\ldots S_p\ x_1\ldots x_n = M\ S_1\ldots S_p\ x_1\ldots x_n$

This property allows to avoid the duplication of environments (through closures) when it is possible to know when the expression whose evaluation is deferred will be evaluated.

Definition 10:

Any expression M such that for all C, $S_1,\ldots,S_m$ there exists some M' such that
$M\ C\ S_1\ \ldots\ S_m = C\ M'$ is called a **stable expression of order m.**

Property 9 allows us to improve the abstraction algorithm by adding a new rule:

(A8$_a$) $[x_1,\ldots,x_n]_p\ M\ N\ =$ **push** $([x_1,\ldots,x_n]_{p-m+1}\ N)\ ([x_1,\ldots,x_n]'_{p+1}\ M)$

if M is a stable expression of order $m \leq p$

The proof of this rule is straightforward:

$M\ N\ S_1\ldots S_p = N\ M'\ S_{m+1}\ldots S_p$ {Def. 10, M is a stable expression of order m}

and:

push $([x_1,\ldots,x_n]_{p-m+1}\ N)\ ([x_1,\ldots,x_n]'_{p+1}\ M)\ S_1\ldots S_p\ x_1\ldots x_n$

$= ([x_1,\ldots,x_n]'_{p+1}\ M)\ ([x_1,\ldots,x_n]_{p-m+1}\ N)\ S_1\ldots S_p\ x_1\ldots x_n$ {definition of **push** }

$= M\ ([x_1,\ldots,x_n]_{p-m+1}\ N)\ S_1\ldots S_p\ x_1\ldots x_n$ {Property 9}

$= ([x_1,\ldots,x_n]_{p-m+1}\ N)\ M'\ S_{m+1}\ldots S_p\ x_1\ldots x_n$ {Def. 10, M is a stable expression of order m}

$= N\ M'\ S_{m+1}\ldots S_p$ {induction hypothesis}

The comparison with **(A8)** shows that rule **(A8$_a$)** allows to avoid the construction of a closure. In intuitive terms, N is known to be the continuation of M, so N will be evaluated with M on the top of the stack. Instead of destroying the environment (in M) and reinstalling it for the evaluation of N, M is abstracted in such a way that it preserves the environment (by []'), so there is no need to save it in a closure.

In order to apply rule **(A8$_a$)** we must detect that an expression is stable of order $m \leq p$. In fact this information can easily be obtained during the first transformation step (compilation of the computation rule). For example property 4 (section 2) allows us to derive:

Property 11:

$(\forall\ E)$ $\mathbb{V}(E)$ is a stable expression of order 0

$(\forall\ E)$ $E = \lambda x.\ E' \Rightarrow (\mathbb{V}(E)\ \mathbf{id})$ is a stable expression of order 1

This property can be exploited to avoid many constructions of closures.

We have described so far the optimizations achieved by improvements of the algorithm itself. We introduce now simplification rules which may be applied to the result of the abstraction algorithm; most of these rules are the translation in the functional framework of well-known compiler optimization techniques called peephole optimizations **[1]**. We give here some of the most useful simplification rules:

(S1)	$\mathbf{dupl_i}\ (\mathbf{move_{i+1,1}}\ E)$	=	$\mathbf{dupl_i}\ E$	
(S2)	$\mathbf{dupl_i}\ (\mathbf{move_{1,j}}\ E)$	=	$\mathbf{dupl_{j-1}}\ E$	
(S3)	$\mathbf{dupl_i}\ (\mathbf{flsh_j}\ E)$	=	$\mathbf{flsh_{j-1}}\ E$	
(S4)	$\mathbf{flsh_i}\ (\mathbf{flsh_j}\ E)$	=	$\mathbf{flsh_{i+j}}\ E$	
(S5)	$\mathbf{flsh_0}\ E$	=	E	
(S6)	$\mathbf{move_{i,j}}\ (\mathbf{flsh_k}\ E)$	=	$\mathbf{flsh_k}\ E$	if $i \leq k$
(S7)	$\mathbf{move_{i,j}}\ (\mathbf{move_{i,k}}\ E)$	=	$\mathbf{move_{i,k}}\ E$	
(S8)	$\mathbf{move_{i,j}}\ (\mathbf{move_{j,i}}\ E)$	=	$\mathbf{move_{i,j}}\ E$	
(S9)	$\mathbf{dupl_i}\ (\mathbf{move_{j,k}}\ E)$	=	$\mathbf{move_{j-1,k-1}}\ (\mathbf{dupl_i}\ E)$	if $j\neq1$, $k\neq1$, $i\neq j-1$
(S10)	$\mathbf{dupl_i}\ (\mathbf{move_{j,1}}\ E)$	=	$\mathbf{move_{j-1,i}}\ (\mathbf{dupl_i}\ E)$	if $j\neq1$
(S11)	$\mathbf{flsh_i}\ (\mathbf{move_{k,l}}\ E)$	=	$\mathbf{move_{k+i,l+i}}\ (\mathbf{flsh_i}\ E)$	
(S12)	$\mathbf{flsh_i}\ (\mathbf{dupl_j}\ E)$	=	$\mathbf{move_{i,i+j}}\ (\mathbf{flsh_{i-1}}\ E)$	

The general idea behind most of these rules is to avoid redundant stack movements. The correctness of each individual rule is obvious. The termination of the rewriting system can be shown using an ordering involving the size of the expression (rules **(S1)** - **(S8)** strictly decrease the size of the expression) weighted by a priority of combinators (rules **(S9)** - **(S12)**).

Let us now come back to the factorial function to illustrate the abstraction algorithm and the simplification rules. The function produced by the first transformation $\mathbb{V}$ is, after simplification (section 3): letrec fact = $\lambda c.\ \lambda x.\ eq_c\ (cond_c\ (c\ 1)\ (sub_c\ (\ fact\ (mult_c\ c\ x))\ x\ 1)\ )\ 0\ x$

Applying the abstraction rules defined above we get:

letrec fact = $[c,x]_0\ (eq_c\ (cond_c\ (c\ 1)\ (sub_c\ (\ fact\ (mult_c\ c\ x))\ x\ 1)\ )\ 0\ x)$

= $\mathbf{dupl_2}\ (\ [c,x]_1\ (eq_c\ (cond_c\ (c\ 1)\ (sub_c\ (\ fact\ (mult_c\ c\ x))\ x\ 1)\ )\ 0))$ **(A2)**

= $\mathbf{dupl_2}\ (\mathbf{push}\ 0\ (\ [c,x]_2\ (eq_c\ (cond_c\ (c\ 1)\ (sub_c\ (\ fact\ (mult_c\ c\ x))\ x\ 1)))))$ **(A4)**

= $\mathbf{dupl_2}\ (\mathbf{push}\ 0\ (\ eq_c\ (\ [c,x]_1\ (cond_c\ (c\ 1)\ (sub_c\ (\ fact\ (mult_c\ c\ x))\ x\ 1)))))$ **(A5)**

= $\mathbf{dupl_2}\ (\mathbf{push}\ 0\ (eq_c\ (cond_c\ ([c,x]_0\ (c\ 1))\ ([c,x]_0\ (sub_c\ (\ fact\ (mult_c\ c\ x))\ x\ 1)))))$ **(A6)**

$([c,x]_0\ (c\ 1))$

= $\mathbf{push}\ 1\ ([c,x]_1\ c)$ **(A4)**

= $\mathbf{push}\ 1\ \mathbf{exed_{1,2,1}}$ **(A7)**

$([c,x]_0\ (sub_c\ (\ fact\ (mult_c\ c\ x))\ x\ 1))$

= $\mathbf{push}\ 1\ ([c,x]_1\ (sub_c\ (\ fact\ (mult_c\ c\ x))\ x\ ))$ **(A4)**

= $\mathbf{push}\ 1\ (\ \mathbf{dupl_3}\ ([c,x]_2\ (sub_c\ (\ fact\ (mult_c\ c\ x)))))$ **(A2)**

= $\mathbf{push}\ 1\ (\ \mathbf{dupl_3}\ (sub_c\ (\ [c,x]_1\ (fact\ (mult_c\ c\ x)))))$ **(A5)**

= $\mathbf{push}\ 1\ (\ \mathbf{dupl_3}\ (sub_c\ (\ \mathbf{push}\ ([c,x]_1\ (mult_c\ c\ x))\ ([x,c]'_2\ fact))))$ **($\mathbf{A8_a}$)**

fact is a stable expression of order 1

= **push** 1 ($\mathbf{dupl}_3$ (sub_c (**push** ($\mathbf{dupl}_3$ ([c,x]$_2$ (mult_c c))) fact))) **(A2),(A8')**

= **push** 1 ($\mathbf{dupl}_3$ (sub_c (**push** ($\mathbf{dupl}_3$ (mult_c ([c,x]$_1$ c))) fact))) **(A5)**

= **push** 1 ($\mathbf{dupl}_3$ (sub_c (**push** ($\mathbf{dupl}_3$ (mult_c $\mathbf{exed}_{1,2,1}$)) fact))) **(A7)**

So the result is:

$\mathbf{dupl}_2$ (**push** 0 (eq_c (cond_c

(**push** 1 $\mathbf{exed}_{1,2,1}$)

(**push** 1 ($\mathbf{dupl}_3$ (sub_c (**push** ($\mathbf{dupl}_3$ (mult_c $\mathbf{exed}_{1,2,1}$)) fact)))))))

The macrocombinator **exed** can be unfolded:

$\mathbf{exed}_{1,2,1}$ = $\mathbf{dupl}_2$ ($\mathbf{move}_{4,2}$ ($\mathbf{move}_{3,1}$ ($\mathbf{flsh}_2$ **id**)))

It can be simplified using the rewriting system **S1-S12**:

$\mathbf{exed}_{1,2,1}$ = $\mathbf{move}_{3,1}$ ($\mathbf{dupl}_2$ ($\mathbf{move}_{3,1}$ ($\mathbf{flsh}_2$ **id**))) **(S9)**

= $\mathbf{move}_{3,1}$ ($\mathbf{dupl}_2$ ($\mathbf{flsh}_2$ **id**)) **(S1)**

= $\mathbf{move}_{3,1}$ ($\mathbf{flsh}_1$ **id**) **(S3)**

And the final version of fact is:

letrec fact = $\mathbf{dupl}_2$ (**push** 0 (eq_c (cond_c

(**push** 1 ($\mathbf{move}_{3,1}$ ($\mathbf{flsh}_1$ **id**)))

(**push** 1 ($\mathbf{dupl}_3$ (sub_c (**push** ($\mathbf{dupl}_3$ (mult_c ($\mathbf{move}_{3,1}$ ($\mathbf{flsh}_1$ **id**))))

fact)))))))

This example illustrates the interest of the simplification strategy: the **dupl** combinators are moved towards the end of the code where they may be cancelled by a **flsh** operator. In operational terms we delay the execution of the **dupl** operators till it becomes really necessary, which may not happen.

The expressions yielded by the abstraction algorithm look very much like machine code. The only remaining difference is that in general these expressions are still binary trees whereas machine code is made of sequences of instructions. The linearization is achieved by the introduction of names denoting embedded composed expressions. In operational terms these names correspond to code addresses. The last remark concerns function names in recursive definitions. These names remain unchanged by the abstraction process since they are free variables. If we assume that names represent code addresses, we must translate the occurrences of function names into **jump** instructions. In functional terms (**jump** f) is defined by **jump** f = f.

The application of the linearization process to the previous expression yields:

letrec fact = $\mathbf{dupl}_2$ (**push** 0 (eq_c (cond_c f_0 f_1)))

let f_0 = **push** 1 ($\mathbf{move}_{3,1}$ ($\mathbf{flsh}_1$ **id**))

let f_1 = **push** 1 ($\mathbf{dupl}_3$ (sub_c (**push** f_2 (**jump** fact))))

let f_2 = $\mathbf{dupl}_3$ (mult_c ($\mathbf{move}_{3,1}$ ($\mathbf{flsh}_1$ **id**)))

We have now several linearized trees which can be written as sequences of instructions in the following way:

```
fact  dupl   2          f1   push   1
      push   0               dupl   3
      eq_c                   sub_c
      cond_c f0, f1          push   f2
f0    push   1               jump   fact
      move   3,1       f2    dupl   3
      flsh   1               mult_c
      id                     move   3,1
                             flsh   1
                             id
```

The appendix describes the evolution of the stack during the execution of (fact **id** 1) and illustrates the duality (functional expression/machine code) of the result of the compilation. The translation of such programs into code for a specific machine is a straightforward macroexpansion. For example, $\mathbf{dupl}_i$ can be expanded into the following 68020 assembly instruction:

```
movl  sp@(4*(i-1)), sp@-        {sp being the stack pointer and 4 the word length}
```

6. CONCLUSION

We advocate in this paper that the whole implementation process of a functional langage should be described in a purely functional way. We have described a method for transforming functions defined in a λ-calculus with constants into "equivalent" functions defined in terms of combinators acting on their arguments like machine instructions on a stack. *The major originality of this approach in contrast with the SECD machine [2,12], the TIM [6] and the CAM [4,15,16] is that we do not have to introduce a machine and describe it in terms of state transitions. The state of the machine is the expression itself and its evolution is specified by the definition of the combinators.* For example, the evaluation of the result of the compilation of the factorial function can be described as the reduction of a functional expression or as the execution of code on a stack machine (see appendix 1). This approach has interesting payoffs as far as correctness proofs are concerned. *We do not have to prove that the operational definition of the machine is coherent with the operational semantics of the language as in [14,17] since they are identical.* The only operational argument in our proof appears in section 2 where it is shown that the reduction of the transformed expression by FIRST amounts to the reduction of the original expression by call-by-value (for instance). However this proof does not involve reasoning on tricky machine states. *We should also mention that the first transformation ($\mathfrak{V}$, $\mathfrak{N}$) does not depend on the chosen*

implementation and could be applied as well in the context of graph reduction (it would lead to a simpler graph evaluator reducing systematically the first term of the expression).

The formalization of the implementation process has also been studied by Reynolds **[19]**, followed by Schmidt **[20]** and Wand **[22]**. They proceed by successive transformations of a semantics of the source language to derive an interpreter or a compiler and an abstract machine. **[22]** presents heuristics for analysing the compilation process. This method also involves continuations and combinators, but in a quite different way: it takes a continuation-based semantics as input whereas in our work continuations appear in the compilation process as a formalization of the computation rule. *Furthermore Wand translates the semantics of the program into a sequence representing the code and a program (or "machine") to execute it; in our approach semantics or machines do not appear explicitly.* In some sense the transformation described in **[22]** to derive machine code can be seen as the specialization of an abstraction algorithm to a particular program which is the semantics of the language. We believe that staying in the functional framework and proceeding exclusively by program transformation (instead of interpreter transformation) makes formal proofs easier. Let us remark however that Wand's goal is a bit different since he deals in the same way with any language (imperative or functional) which can be described by a continuation semantics.

The benefits of continuations to compiler design has already been illustrated by work on Orbit **[11]**. This project integrates continuation conversion as a preliminary "standardization" step but the compilation is not entirely described by program transformation. This work puts deliberate emphasis on efficiency issues rather than on the correctness of the implementation.

Even if the performance considerations are not the main topic of this paper we have to say a few words about the produced code. We have chosen the environment-based approach rather than graph reduction because it is closer to traditional von Neumann machines. The code produced by our transformation rules is lower level than the code of **[2]** for the SECD machine and **[4,16]** for the CAM. In operational terms the main difference with the SECD machine is the place where environment savings are achieved: we achieve the savings when encountering intermediate subexpressions rather than at function call. We depart from the CAM as far as environment representation is concerned ; in the CAM, environments are represented as trees which entails less expensive closure building but costly access time. Another possible choice is to keep a pointer to the global environment and to distinguish access to local identifiers and to global identifiers **[3]**; this makes function calls more efficient because context switching amounts to a pointer movement.

A prototype compiler based on this approach has been implemented on a SUN workstation. The first transformation includes three computation rules: call-by-value, call-by-name and call-by-need. The expression of call-by-need in the functional framework has not been presented here: it requires the addition of a new argument (an array representing the memory) to expressions. The interested reader may refer to **[7]** for a full treatment of call-by-need. Let us point out that the implementation

of a compiler based on our transformation rules in a language with pattern matching such as ML is quite straightforward. The following table shows the execution times of the code produced by our compiler for the traditional fib 20 with call-by-name, call-by-value and call-by-need.

	c. b. value	c. b. name	c. b. need
time	80 ms	2.2 s	0.55 s
call/sec	274 000	10 000	40 000

We should mention that only the simple language described in section 1 has been implemented so far and these performances have to be confirmed for a more realistic language including lists, user-defined data types and pattern matching. Nevertheless these results are promising because they show that the code produced by our compiler for simple programs is realistic (the efficiency of the code produced by the C compiler for the same function is 263,000 call/sec). *These performance are made possible by the program transformation approach which allows the systematic application of optimization rules*. Each technique is applied at the appropriate transformation level; for example tail-recursion optimization is achieved after the first transformation step whereas peephole optimizations are applied on the resulting code. Actually most well-known compiler optimization techniques (common subexpression elimination, stack pollution,...) can be described in a functional way.

One of the promises of our approach could be the expression within the same framework of various implementation techniques. This would provide a better understanding of the relations between the different implementations which is very difficult to assess in general.

REFERENCES

1. AHO, A.V., SETHI, R. and ULLMAN, J. D. *Compilers: Principles, Techniques and Tools*. Addison-Wesley 1986.
2. BURGE, W. H. *Recursive Programming Techniques*. Addison-Wesley, 1975.
3. CARDELLI, L. Compiling a functional language. In *Proceedings of 1984 ACM Symposium on Lisp and Functional Programming* (1984), 208-226.
4. COUSINEAU, G., CURIEN, P.-L., and MAUNY, M. The categorical abstract machine. *Science of Computer Programming 8* (1987), pp.173-202.
5. CURRY, H. B., FEYS, R., and CRAIG, W. *Combinatory Logic. Vol. I*, North-Holland, (1958, Second printing 1968).

6. FAIRBAIRN, J., and WRAY, S. C. TIM: a simple abstract machine for executing supercombinators. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture. Lecture Notes in Computer Science 274* (1987), Springer, Berlin, 34-45.

7. FRADET, P. Compilation des langages fonctionnels par transformation de programmes. Thèse de Doctorat de l'Université de Rennes I, (Nov. 1988).

8. HUGHES, R.J.M. Supercombinators: a new implementation method for applicative languages. In *Proceedings of 1982 ACM Symposium on Lisp and Functional Programming* (1982), 1-10.

9. JOHNSSON, T. Efficient compilation of lazy evaluation.In *Proceedings of 1984 ACM SIGPLAN Symposium on Compiler Construction, SIGLAN Notices 19, 6,*(1984), 58-69.

10. JOHNSSON, T. Target code generation from G-Machine code. In *Proceedings of the Workshop on Graph Reduction, Lecture Notes in Computer Science 279* (1986), Springer, Berlin, 119-159.

11. KRANZ, D., KELSEY, R., REES, J., HUDAK, P., PHILBIN, J., and ADAMS, N. Orbit: an optimizing compiler for scheme. In *Proceedings of 1986 ACM SIGPLAN Symposium on Compiler Construction,* (1986), 219-233.

12. LANDIN, P.J. The mechanical evaluation of expressions. *Computer Journal 6* (1964), 308-320.

13. LEMAITRE, M., CASTAN, M., DURAND, M.-H., DURRIEU, G., and LECUSSAN, B. Mechanisms for Efficient Multiprocessor Combinator Reduction. In *Proceedings of 1986 ACM Symposium on Lisp and Functional Programming* (1986), 113-121.

14. LESTER., D. The G-Machine as a representation of stack semantics. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture. Lecture Notes in Computer Science 274* (1987), Springer, Berlin, 46-59.

15. MAUNY, M. Compilation des langages fonctionnels dans les combinateurs categoriques. Application au langage ML. Thèse de Troisième Cycle de l'Université de Paris VII, 1985.

16. MAUNY, M., and SUAREZ, A. Implementing functional languages in the categorical abstract machine. In *Proceedings of 1986 ACM Symposium on Lisp and Functional Programming* (1986), 266-278.

17. PLOTKIN, G. D. Call-by-name, call-by-value and the λ-Calculus. *Theoretical Computer Science 1,* (1975), 125-159.

18. PLOTKIN, G. D. A structural approach to operational semantics. University of Aarhus, DAIMI FN-19 (1981).

19. REYNOLDS, J. C. Definitional interpreters for higher-order programming languages. In *Proceedings of ACM Annual Conference 2* (1972), 717-740.

20. SCHMIDT, D. A. State transition machines for lambda-calculus expressions. In *Proceedings of Semantics Directed Compiler Generation Workshop, Lecture Notes in Computer Science 94* (1980), Springer, Berlin, 415-440.

21. TURNER, D. A. A new implementation technique for applicative languages. *Software-Practice and Experience 9* (1979), 31-49.

22. WAND, M. Deriving target code as a representation of continuation semantics. *ACM Trans. on Program. Lang. and Systems 4, 3* (1982), 496-517.

APPENDIX

We describe the evaluation of the result of the compilation of fact given at the end of section 5. The function is applied to **id** 1 since the new definition of fact takes two arguments: a continuation which is equal to **id** and an integer value. We show in parallel the evaluation as the execution of code on a stack machine and as the call-by-name reduction of a functional expression. In the left part of the figure (Machine Code) we represent the state of the stack (the top being the leftmost element) before the execution of the corresponding instruction.

	Machine Code		**Functional Expression**
	code	**stack**	
fact	dupl 2	id:1	**dupl**$_2$ (**push** 0 (eq$_c$ (cond$_c$ f$_0$ f$_1$))) **id** 1
	push 0	1: id:1	**push** 0 (eq$_c$ (cond$_c$ f$_0$ f$_1$)) 1 **id** 1
	eq$_c$	0:1: id:1	eq$_c$ (cond$_c$ f$_0$ f$_1$) 0 1 **id** 1
	cond$_c$ f0, f1	False: id:1	cond$_c$ f$_0$ f$_1$ False **id** 1
f1	push 1	id:1	**push** 1 (**dupl**$_3$ (sub$_c$ (**push** f$_2$ (**jump** fact)))) **id** 1
	dupl 3	1:id:1	**dupl**$_3$ (sub$_c$ (**push** f$_2$ (**jump** fact))) 1 **id** 1
	sub$_c$	1:1:id:1	sub$_c$ (**push** f$_2$ (**jump** fact)) 1 1 **id** 1
	push f2	0:id:1	**push** f$_2$ (**jump** fact) 0 **id** 1
	jump fact	f2:0:id:1	**jump** fact f$_2$ 0 **id** 1
fact	dupl 2	f2:0:id:1	**dupl**$_2$ (**push** 0 (eq$_c$ (cond$_c$ f$_0$ f$_1$))) f$_2$ 0 **id** 1
	push 0	0:f2:0:id:1	**push** 0 (eq$_c$ (cond$_c$ f$_0$ f$_1$)) 0 f$_2$ 0 **id** 1
	eq$_c$	0:0:f2:0:id:1	eq$_c$ (cond$_c$ f$_0$ f$_1$) 0 0 f$_2$ 0 **id** 1
	cond$_c$ f0, f1	True:f2:0:id:1	cond$_c$ f$_0$ f$_1$ True f$_2$ 0 **id** 1
f0	push 1	f2:0:id:1	**push** 1 (**move**$_{3,1}$ (**flsh**$_1$ **id**)) f$_2$ 0 **id** 1
	move 3,1	1:f2:0:id:1	**move**$_{3,1}$ (**flsh**$_1$ **id**) 1 f$_2$ 0 **id** 1
	flsh 1	1: f2:1:id:1	**flsh**$_1$ **id** 1 f$_2$ 1 **id** 1
	id	f2:1:id:1	**id** f$_2$ 1 **id** 1
f2	dupl 3	1:id:1	**dupl**$_3$ (mult$_c$ (**move**$_{3,1}$ (**flsh**$_1$ **id**))) 1 **id** 1
	mult$_c$	1: 1:id:1	mult$_c$ (**move**$_{3,1}$ (**flsh**$_1$ **id**)) 1 1 **id** 1
	move 3,1	1:id:1	**move**$_{3,1}$ (**flsh**$_1$ **id**) 1 **id** 1
	flsh 1	1: id:1	**flsh**$_1$ **id** 1 **id** 1
	id	id:1	**id id** 1
	id	1	**id** 1
	result = 1		**1**

LISTE DES DERNIERES PUBLICATIONS INTERNES

PI 459 **SYNCHRONOUS PROGRAMMING WITH EVENTS AND RELATIONS : THE SIGNAL LANGUAGE AND ITS SEMANTICS**
Albert BENVENISTE, Paul LE GUERNIC
66 Pages, Février 1989.

PI 460 **FLOW ANALYSIS IN TANDEM QUEUES WITH FEEDFORWARD FLOWS**
Kamel SISMAIL
14 Pages, Février 1989.

PI 461 **NUMERICAL CONCERNS IN CONVOLUTION-TYPE ALGORITHMS**
Gerardo RUBINO, William STEWART
18 Pages, Février 1989.

PI 462 **ACCUMULATED REWARD OVER THE N FIRST OPERATIONAL PERIODS IN FAULT-TOLERANT COMPUTING SYSTEMS**
Gerardo RUBINO, Bruno SERICOLA
14 Pages, Mars 1989.

PI 463 **ELEMENTS FINIS C^1, POLYNOMIAUX DE DEGRE QUATRE PAR TRIANGLE, DANS UNE TRIANGULATION FORMEE DE TRIANGLES EQUILATERAUX**
Michel CROUZEIX, Miloud SADKANE
26 Pages, Mars 1989.

PI 464 **VERS UN MODELE D'ECLAIREMENT REALISTE**
Pierre TELLIER, Kadi BOUATOUCH
66 Pages, Avril 1989.

PI 465 **COMPILATION OF FUNCTIONAL LANGUAGES BY PROGRAM TRANSFORMATION**
Pascal FRADET, Daniel LE METAYER
28 Pages, Avril 1989.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique